Inplace Run-Length 2d Compressed Search

Amihood Amir* Gad M. Landau† Dina Sokol‡

Abstract

The recent explosion in the amount of stored data has necessitated the storage and transmission of data in compressed form. The need to quickly access this data has given rise to a new paradigm in searching, that of *compressed matching* [1, 8, 10]. The goal of the compressed pattern matching problem is to find a pattern in a text *without decompressing the text*.

The criterion of extra space is very relevant to compressed searching. An algorithm is called *inplace* if the amount of extra space used is proportional to the input size of the pattern. In this paper we present a 2d compressed matching algorithm that is inplace. Let $compressed(T)$ and $compressed(P)$ denote the compressed text and pattern, respectively. The algorithm presented in this paper runs in time $O(|compressed(T)| + |P| \log \sigma)$ where σ is $\min(|P|, |\Sigma|)$, and Σ is the alphabet, for all patterns that have no trivial rows (rows consisting of a single repeating symbol). The amount of space used is $O(|compressed(P)|)$. The compression used is the 2d run-length compression, used in FAX transmission.

1 Introduction

As technology develops in diverse areas, from medicine to multimedia, there is a continuous increase in the amount of stored digital data. This increase has made it critically important to store and transmit files in a compressed form. The need to quickly access this data has given rise to a new paradigm in searching, that of *compressed matching* [1, 8, 10]. In traditional pattern matching, the pattern (P) and text (T) are explicitly given, and all occurrences of P in T are sought. In compressed pattern matching the goal is the same, however, the pattern and text are given in compressed form. Let *compress* be a compression algorithm, and let $compressed(D)$ be the result of *compress* compressing data D. A compressed matching algorithm has been defined to be *optimal* if its search time is $O(|compressed(T)|)$ [5]. The *compression ratio* of a file is the ratio of the size of the uncompressed file to the size of the compressed file. It is important to note that in a case where the compression ratio of the given text is a constant, an optimal compressed matching performs

*Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8770, and College of Computing, Georgia Tech; `amir@cs.biu.ac.il`; Partially supported by NSF grant CCR-01-04494 and ISF grant 282/01.

†Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, phone: (718) 260-3154, FAX: (718) 260-3906; email: `landau@poly.edu`; partially supported by NSF grants CCR-9610238 and CCR-0104307, by NATO Science Programme grant PST.CLG.977017, by the Israel Science Foundation (grants 173/98 and 282/01), and by IBM Faculty Partnership Award.

‡Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; `sokold@cs.biu.ac.il`

no better than the naive algorithm of decompressing the text. However, if the constants hidden in the *"big O"* are smaller than the compression ratio, then the compressed matching does offer a practical advantage.

Although optimality in terms of time is always important, when dealing with compression, the criterion of **extra space** is perhaps more important [15]. Applications employ compression techniques specifically because there is a limited amount of available space. Thus, it is not sufficient for a compressed matching algorithm to be optimal in terms of time, but it must also satisfy the given space constraints. Space constraints may be due to limited amount of disk space, (e.g. on a server), or they may be related to the size of the memory or cache. Note that if an algorithm uses as little extra space as the size of the cache, the runtime of the algorithm is also greatly reduced as no cache misses will occur [11].

Definition 1 (inplace) *We say that a compressed matching is* inplace *if the extra space used is proportional to the input size of the pattern.*

Note that this definition encompasses the compressed matching model (e.g. [2]) where the pattern is input in uncompressed form, as well as the fully compressed model (e.g. [7]), where the pattern is input in compressed form. The *inplace* requirement allows the extra space to be the input size of the pattern, whatever that size may be.

In this paper, we present an *inplace* algorithm to solve the two dimensional compressed matching problem for the run-length compression (used by FAX transmission) when both the input text and pattern are compressed. The extra space used is $O(|compressed(P)|)$. This space constraint is smaller than that of all known compressed matching algorithms. In addition, our algorithm has optimal search time.

Amir, Benson and Farach [5] presented an algorithm that solved this problem in *both* time and space $O(|compressed(T)| + |P|)$[1]. Using known techniques, it is possible to modify their algorithm so that its extra space is $O(|P|)$. However, using the run-length compression, the difference between $|P|$ and $|compressed(P)|$ can be quadratic. Moreover, the constants involved in the complexity of their algorithm are large, while our constants are relatively small.

In [6] we presented an inplace algorithm for a similar problem. There, the definition of a match was restricted in the sense that a pattern occurrence had to be sharp, i.e. it could not blend in with its surroundings. In this paper we solve the more general problem of finding every occurrence of the pattern in the text using the traditional definition of a pattern match.

In the following section we give a formal definition of the problem and a high-level description of the algorithm. Sections 3 and 4 describe the algorithm in detail. In Section 5 we discuss patterns that contain trivial rows.

[1]The original algorithm of ABF takes time $O(|compressed(T)| + |P| \log \sigma)$, where σ is $\min(|P|, |\Sigma|)$, and Σ is the alphabet. However with the witness table construction of Galil and Park [9] the time is reduced to $O(|compressed(T)| + |P|)$.

2 Preliminaries

2.1 Problem description

Let $S = s_1 s_2 \ldots s_n$ be a string over some alphabet Σ. The *run-length compression* of string S is the string $S' = \alpha_1^{r_1} \alpha_2^{r_2} \ldots \alpha_k^{r_k}$ such that: (1) $\alpha_i \neq \alpha_{i+1}$ for $1 \leq i < k$; and (2) S can be described as the concatenation of k *segments*, the symbol α_1 repeated r_1 times, the symbol α_2 repeated r_2 times, $\ldots$, and the symbol α_k repeated r_k times. The *two-dimensional run length compression* is the concatenation of the run-length compression of all the matrix rows.

We define the **Two-Dimensional Run-Length Compressed Matching Problem** as follows:

Input: Text array T of size $n \times n$, and pattern array P of size $m \times m$ *both* in Two-dimensional run-length compressed form.

Output: All locations in T of occurrences of P. Formally, the output is the set of locations (i, j) such that $T[i+k, j+l] = P[k+1, l+1]\ \ k, l = 0 \ldots m-1$.

Definition 2 (trivial) *A row of the pattern is called* trivial *if every location on the row contains the same character.*

In this paper we describe an algorithm for patterns that do not contain any trivial rows. The algorithm is inplace and runs in time $O(|compressed(T)| + |P| \log \sigma)$, where σ is $\min(|P|, |\Sigma|)$. Patterns that contain trivial rows introduce difficulties for compressed matching. In Section 5 we address these issues and we show how our algorithm can be modified to find all patterns. The modified algorithm is inplace, however, its search time is $O(|T|)$.

2.2 Algorithm Idea

Our algorithm uses the framework of the non-compressed 2-D pattern matching algorithm of [4]. The idea of this algorithm is to use the **dueling** mechanism defined by Vishkin [14]. Applying the dueling paradigm directly to run-length compressed matching has previously been considered impossible. In [5] it has been stated that the dueling paradigm cannot be directly applied to compressed matching since the location of a witness in the compressed text cannot be accessed in constant time. In this paper we show a way in which a witness *can* be accessed in (amortized) constant time, enabling a relatively straightforward application of the dueling paradigm to compressed matching. Following is a brief description of the algorithm of [4].

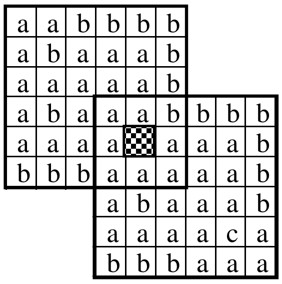
2.2.1 Pattern Preprocessing

A witness table, $Witness[1 \ldots m, 1 \ldots m]$, is built for the $m \times m$ pattern P. Given 2 overlapping copies of a pattern, a *witness* is a location at which a conflict occurs. $Witness[i, j]$ is the position of one such conflict when the pattern is placed upon itself at position (i, j). If all overlapping elements are the same, then we set $Witness[i, j] = *$. See figure 1 for an example of a witness.

The witness table is constructed in $O(m^2 \log \sigma)$ time. The witness table construction uses the algorithm of Main and Lorentz [13] which finds the longest prefix of a pattern string occurring

a	a	b	b	b	b
a	b	a	a	a	b
a	a	a	a	a	b
a	b	a	a	a	b
a	a	a	a	c	a
b	b	b	a	a	a

(a)



(b)

	1	2	3	4	5	6
1	*	1,3	1,3	1,4	1,5	1,6
2	2,2	2,2	2,5	3,5	2,6	2,6
3	3,3	3,4	3,5	4,5	3,6	3,6
4	4,2	4,2	4,5	5,5	4,6	4,6
5	5,3	5,4	5,5	5,5	5,5	*
6	6,1	6,2	6,6	6,4	*	*

(c)

Figure 1: (a) A pattern P, which will be used throughout the paper. (b) A sample witness. 2 overlapping copies of P are shown. The position of a conflict is shaded, setting Witness[4,4]=(5,5). (c) The complete witness table for P.

at each position of a text string. In order to use the Main and Lorentz algorithm (ML), we must convert the 2d problem into a problem on strings, which is done as follows. The pattern is processed column-by-column, and we describe the processing of column j. The text string input to ML is the pattern block $P[1 \ldots m,\ j \ldots m]$, where each row's suffix from column j through m is viewed as a single character. Similarly, the pattern input to ML is the pattern block $P[1 \ldots m,\ 1 \ldots m-j+1]$, where each row's prefix is viewed as a single character. The algorithm of ML then finds, for each location on column j of the pattern, the longest match of the pattern itself. If the match extends down to row m, there is no witness. Otherwise, we simply locate the position of the mismatch to obtain the witness.

A suffix tree of the pattern rows is built, and it is processed for LCA queries, to enable constant time comparisons of substrings.

2.2.2 Text Scanning

Part 1 (Candidate Consistency): A *candidate* is a location in the text where the pattern may occur. We say that two candidates are *consistent* if they expect the same text characters in their region of overlap (*i.e.* the value of the witness is a $*$). At the start of the candidate consistency step, every text location is a potential candidate. The goal of the candidate consistency step is to eliminate candidates by performing duels, until all remaining candidates are mutually consistent.

A duel is performed between 2 candidates as follows. We first compute the distance between the candidates, and access the witness table at that position. If the value of the witness is a *, then both candidates remain alive. Otherwise, the value of the text location at the witness position is compared with the witness, and the appropriate candidate is killed. As an example, we refer back to the witness shown in figure 1b. Assume that we are searching a text for the pattern shown in figure 1a, and there are 2 candidates in the text, one at location (r, c) and one at $(r+3, c+3)$. To duel, we check whether the character in the text at location $(r+4, c+4)$ is a 'c.' If it is, then the second candidate is killed, otherwise, the first is killed. Note that if the character is incorrect for both candidates (in this case, it is neither a 'c' nor a 'b'), then both candidates can be killed. However, this modification does not change the time complexity of the algorithm.

In [4] a method was shown for performing all of the duels in linear time. The order of the duels is as follows. First, a 1-D algorithm is applied to each column of the text. Upon completion, all candidates within each column are consistent. The 2-D algorithm starts with the rightmost column and adds one column at a time, from right to left. As column i is added, the following invariant is maintained: all candidates in columns $i+1 \ldots n$ are pairwise consistent. Because of the transitivity of the consistency relation, when adding column i, it is necessary to perform at most one duel per row in which both candidates remain alive. Thus, for a text of size n^2 the addition of each column takes $O(n)$ time, and the overall time of the dueling stage is $O(n^2)$.

Part 2 (Verification): In the verification phase the text elements are compared to the pattern elements to discover actual occurrences of the pattern. Although a given text location may be contained in several candidates, since all of the candidates are mutually consistent, each text element must only be compared to a single pattern element.

The verification phase consists of two *waves*, a forward wave and a backward wave. The forward wave is done first vertically, and then horizontally. The vertical wave moves down the text columns, verifying the first column of each candidate occurrence, while the horizontal wave moves along the rows of the text, comparing pattern elements with text elements. Following the forward wave, and in a similar manner, the backward wave is used to discard all candidates that contain one or more mismatches. Upon completion, all remaining candidates are pattern occurrences.

3 Pattern Preprocessing

The critical tool used in performing a duel is a witness table of the pattern. Our algorithm uses a witness table that is conceptually similar to the one described in the previous section, however, it is constructed according to the compressed pattern. In this section we describe our witness table in detail and show how it can be constructed in $O(m^2 \log \sigma)$ time.

3.1 Characteristics of the Witness Table

Storing witnesses according to $compressed(P)$ is the central theme of our algorithm. It helps obtain the following 2 goals: (a) it keeps the extra space $O(|compressed(P)|)$, (b) it enables fast access of the witness location in the compressed text. Lemmas 1 and 2 (resp.) show how our witness table achieves these 2 goals.

Definition 3 (implicit witness) *An* implicit witness *is a witness that is easy to compute from other entries in the witness table.*

Lemma 1 *It is enough to store* at most *1 witness for each compressed character in $compressed(P)$.*

Proof: Part 1 of the proof deals with all compressed characters in the pattern besides the last compressed character in each row. Part 2 deals with the entries in the last compressed column.

Part 1: For each element in $compressed(P)$, excluding the last compressed character in each row, there is only one possible position at which a new occurrence of the pattern can start. This position is the place within the compressed character at which the character (0,0) of *compressed*(P) occurs. For all locations besides for this one position, position $(0,0)$ of the second pattern is an implicit witness.

Part 2: For the last compressed character on each row one witness is not enough, since location (0,0) of *compressed*(P) can begin in numerous places, continuing past the end of the pattern. There may be up to $O(m)$ possible places of overlap. For example, consider the character a^3 at location $(6,2)$ of the pattern in figure 2a. It is possible for an overlapping pattern to occur either at the $2nd$ or the $3rd$ 'a' within the run. A separate witness would be necessary for each of these possibilities. We show that no more than $|compressed(P)|$ explicit witnesses are actually necessary.

As we will explain, our algorithm works with pieces of the text of size $3m/2 \times 3m/2$. The candidates all lie in the upper-left $m/2 \times m/2$ square (*quadrant*). Any two candidates that are dueling will have row/column distance at most $m/2$. Thus, we only have to store witnesses for the upper-left quadrant of the pattern. We consider the following 3 possibilities.

Case 1: $compressed(P)[0,0] = a^k | k < m/2$
The same argument as in Part 1 of the proof is true here since position $(0,0)$ must fit exactly into any compressed location that starts before column $m/2$.

Case 2: $compressed(P)[i,0] = a^k | k < m/2$ for some row $0 \leq i < m/2$
This is similar to case 1, however, we use location $(i,0)$ (of the second pattern) as the implicit witness. The explicit witnesses for a location in row r are stored in row $r+i$ since at most one explicit witness is necessary for each appearance of the character $(i,0)$ at row $r+i$ of the pattern.

Case 3: $compressed(P)[i,0] = a^k | k \geq m/2$ for all rows $0 \leq i < m/2$
Since there are no trivial rows, every compressed character in the upper-left quadrant is the first in its row, which is by definition not the last.

□

There are two important characteristics that are unique to our witness table. In Lemma 2 we show how these properties enable constant access of the witness in the compressed text.

1. Within the row of the witness, the witness location stored is the **rightmost** conflict in the overlap. (Note that the row of the witness is arbitrary.)
2. The position of the witness is stored as the compressed offset (i.e. the number of compressed characters) from the **end** of overlapping region.

Consider the sample uncompressed witness shown in figure 1b. The compressed witness corresponding to it is $Witness[4,3] = (5,-2)$. This indicates that the compressed location of the 'c' is in row 5, and it is the 2nd compressed character from the right. In figure 2 we show the complete witness table for the compressed version of the pattern shown in figure 1a.

Lemma 2 *If the witness location is the* rightmost *conflict in its row, and it is stored as the compressed offset from the* end *of the overlap, then it is possible to access the witness in the text in constant time.*

a^2	b^4		
a	b	a^3	b
a^5	b		
a	b	a^3	b
a^4	c	a	
b^3	a^3		

(a)

*			
		3,-1	
4,-1			
		5,-2	
5,-1			

(b)

0	2		
0	1	2	5
0	5		
0	1	2	5
0	4	5	
0	3		

(c)

Figure 2: (a) *compressed*(P) for the pattern shown in figure 1a. (b) The witness table for *compressed*(P). The negative numbers indicate an offset from the end of the pattern. Each empty location (i, j) indicates that position (i, j) itself is the witness. (c) The cumulative run-lengths of *compressed*(P).

Proof: Consider 2 candidates performing a duel, UL (for upper-left) and BR (for bottom right). The row of the witness in the compressed text is depicted in figure 3. We assume that we have a pointer to the right end of UL, (index j) which is the end of the overlapping region (we show later that this is possible). If we offset the witness from point j, then only the candidate that contains an error will be killed.

Case 1: The text mismatches the patterns in the area between the witness and the end of the overlap. In this case *both* candidates mismatch the text since the candidates expect the same characters in the area following the witness. Therefore, either candidate (or both) can be justifiably killed.

Case 2: The text matches the patterns between the witness and the end of the overlap. Since both candidates expect the same elements, and the text agrees, the witness accessed will be exactly at the proper location. The candidate in error will be killed.

□

Note that Lemma 2 does not hold if the witness would be stored as an offset from the left edge of UL (location i). In this case, the text corresponding to the nonoverlapping prefix of UL (indices $i \dots k$ in figure 3) affects the location of the witness. If the run-lengths in the text do not match those of the pattern, the witness retrieved may not even lie in the candidates' overlap. However, an alternative witness would be the leftmost conflict in the overlap as an offset from the left edge of BR (index k). We chose the right edge of UL since it is easy to have the pointer at the proper location in the text.

In addition to the witness table, we store the cumulative run-lengths up until each position of *compressed*(P) (figure 2c).

3.2 Building the Witness Table

In [4] the witness table is built in time $O(m^2 \log \sigma)$ using the algorithm described in the previous section. Galil and Park [9] presented a linear time, $O(m^2)$, algorithm for constructing a witness table of an $m \times m$ pattern. Their algorithm is a recursive algorithm which analyzes the periodicity of subpatterns. Unfortunately, we were unable to use this algorithm for compressed matching and still satisfy our space constraints and our specific characteristics of the witness table (described in the previous subsection). However, we are able to directly apply the algorithm of [4], keeping the space $O(|compressed(P)|)$. We use the exact algorithm of [4], however, we build the suffix tree of

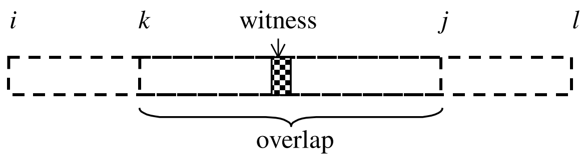


Figure 3: The row of the witness in the compressed text is depicted. Indices i and j mark the beginning and end of the upper-left pattern UL, and $k \dots l$ mark the bottom-right pattern BR. The witness is the *rightmost* conflict in the overlap, therefore the text either matches or mismatches *both* patterns in the area between the witness and the right end of the overlap.

the rows of the compressed pattern. Each compressed character in the pattern defines 2 nodes in the suffix tree, one for its character, and one for its length.

We process each logical column which may need a witness, and store only the necessary witnesses, of which there are $O(|compressed(P)|)$ (by Lemma 1). Each time a substring comparison is done, we simply use the suffix tree of the compressed pattern, and check the edges separately. Since we need the rightmost mismatch on the witness row, we must build an additional suffix tree of the rows from right to left to use when finding the location of the witness. Algorithm A describes the construction of the witness table.

Algorithm A: Pattern Preprocessing

Input: 2-D pattern in run-length compressed form, called $compressed(P)$.
$compressed(P)[0,0] = a^{k_1}, a \in \Sigma, 1 \leq k_1 < m$.
Output: a witness table, with size $O(|compressed(P)|)$.

Begin Algorithm
// Step A.1:
Build a suffix tree of the rows of $compressed(P)$. (Each element in $compressed(P)$ defines 2 nodes in the suffix tree, one for its character, and one for its length.)
Preprocess the suffix tree for LCA queries.
Do the same for the rows in reverse order.

// Step A.2: Goal: Mark each position of $compressed(P)$ that needs an explicit witness.

Case 1: $k_1 < m/2$
For $0 \leq i < m/2$ mark each position in row i with logical column $0 \leq j < m/2$ which contains an element $a^l | l \geq k_1$.
Case 2: $k_1 > m/2$ AND for some row $0 \leq i < m/2$ $compressed(P)[i,0] = b^{k_2} | b \in \Sigma, k_2 < m/2$
Mark each position that contains an element $b^l | l \geq k_2$.
Case 3: $k_1 \geq m/2$ for all rows $0 \leq i < m/2$
For $0 \leq i < m/2$ mark each position in row i with logical column $0 \leq j < m/2$ which contains an element $a^l | l \geq k_1$.

// Step A.3: Goal: Compute the location of a witness for each location marked in step A.2.

Allocate a set of m pointers to the rows of $compressed(P)$, initialize each to point to the end of the row.
For $j = m/2 - 1$ down to 0,
if column j has at least 1 marked location
Move the pointers (from right to left) to logical column j of $compressed(P)$ and apply the ABF pattern preprocessing algorithm as described in section 2.2.1.
The suffix trees and LCA enable constant time comparison of the compressed substrings excluding the first and last element. The edges must be checked separately.

// Step A.4: Goal: Ensure that the witness stored is the rightmost in its row.

Reset the m pointers to point to the end of the rows of $compressed(P)$.
For $j = m/2 - 1$ down to 0,
if column j has at least 1 marked location
Move the pointers (from right to left) to logical column $m - j + 1$ of $compressed(P)$.
For each logical location (r, j) that has a witness, use the reverse suffix tree in the row of the witness to find the location of the rightmost conflict. For cases 1 and 3, the witness computed for a given location is stored at its compressed location in the pattern. For case 2 patterns, the witness computed for the uncompressed location (r, j) is stored in the compressed location of row $r + i$ with logical column j, where i is as defined in Step A.2 Case 2.

End Algorithm

Complexity: The runtime of the algorithm is $O(m^2 \log \sigma)$ as in the uncompressed version. The suffix trees and the witness table both use $O(|compressed(P)|)$ space.

4 Text Scanning

4.1 Framework

In order to minimize our extra space, the text scanning works with overlapping pieces of the text. Each compressed text block corresponds to a $3m/2 \times 3m/2$ square of uncompressed text. Since the compression is row-by-row, each block contains exactly $3m/2$ rows, while the lengths of the rows vary according to the text. The potential candidates all lie in the upper-left $m/2 \times m/2$ square.

We divide the text into blocks with $3m/2$ logical rows/columns as follows. The current block is represented by $3m$ pointers, 2 per row. On a given row, the first pointer points to the left edge of the block and the second pointer points to the right end of the block. The logical distance between the 2 pointers is $3m/2$. To move from one text block to the next, we simply move the pointers sequentially to the right. However, we must make sure that each text character is included in at most 2 different blocks. Since the pattern contains no trivial rows, we can skip all text characters that have runs longer than m. For example, say that row m of the text begins with the character a^{10m}. We can immediately move the pointers in rows 1 through m to column $9m$, skipping the first 9 text blocks. Similarly, if row $m+1$ begins with a^{10m} then all rows below it are not relevant until the $9th$ block. Thus, we begin moving the pointer in the mth row, and then continue with rows $m-1 \ldots 1$ followed by rows $m+1 \ldots 3m/2$.

4.2 Candidate Consistency (Dueling)

4.2.1 Marking Potential Candidates

Observation 1 *Every compressed text element contains at most 1 possible pattern start.*

The first step of the candidate consistency phase is to mark the potential candidates that should be involved in the dueling phase. Following Observation 1, the number of potential candidates at the start of the dueling phase is no more than the size of the compressed text. However, even this number may be too large since $|compressed(T)|$ may be greater than $|compressed(P)|$. In order to remain *inplace* it is important that we never keep a list of size greater than $O(|compressed(P)|)$. To overcome this problem, in a case where $|compressed(T)| > |compressed(P)|$, the candidate consistency phase begins with a simple procedure that reduces the number of candidates from $|compressed(T)|$ to $|compressed(P)|$. The procedure performs a linear-time search of the text for the *shortest* compressed pattern row. Each such occurrence defines exactly one possible pattern start. To search for a 1-D run-length compressed string, the Knuth-Morris-Pratt [12] algorithm can be used to find all exact occurrences of the string without its first and last characters. We then have to check whether the edges fit on either end.

Lemma 3 *The number of occurrences of the shortest compressed pattern row is no more than* $|compressed(P)|$.

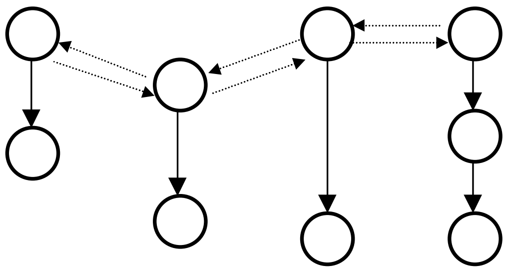


Figure 4: The linked list of potential candidates at the start of the candidate consistency phase, sorted first within each logical column, and then between columns. The size of the list is $\leq |compressed(P)|$.

Proof: Let r be the shortest row of $compressed(P)$. Let ℓ be the compressed length of row r and c be the compressed length of the period of r. Note that $c \geq 2$ and that $\ell = c$ if row r is not periodic.

Within a given text row, there are at most ℓ/c overlapping occurrences of row r. Since we search $m/2$ rows, overall there are no more than $\frac{m}{2} \cdot \frac{\ell}{c}$ starts of row r. $\frac{m}{2} \cdot \frac{\ell}{c} < m\ell \leq |compressed(P)|$. □

Corollary: The number of potential candidates at the start of the candidate consistency phase is at most $|compressed(P)|$.

The output of the module is a linked list of candidates. Each candidate is defined by a triple (r, c, c'), where (r, c) is the candidate's location in the compressed text, and c' is its *logical* column, i.e. its column in the uncompressed text. As the candidates are found, row by row, they are sorted by their logical columns (c') using the bucket sort. We maintain m buckets, and each candidate is placed in the proper bucket according to c'. Upon completion, each bucket contains a linked list of candidates that occur in its logical column, sorted by row. We link the individual buckets that contain candidates both from left to right and from right to left (see figure 4), and we discard the empty buckets.

4.2.2 Dueling Order

Dueling between all candidates is necessary to ensure that the candidates are pairwise consistent. Dueling between all pairs in the naive way may result in $O(|compressed(T)|^2)$ duels. In order to remain optimal, we would like the number of duels to be proportional to the size of the compressed text. In this subsection we describe how this is possible. In the following subsection we describe how the actual duel is performed on the compressed text.

We consider the four periodicity classes defined in [3]. We show separately for each periodicity type that no more than $O(|compressed(T)|)$ duels will be necessary. Following we describe, for each periodicity class, the way the pattern starts may appear in an $m/2 \times m/2$ text block B.

1. Non-periodic: There is at most one pattern occurrence in B.

2. Line Periodic: The pattern starts in B all fall on one line.

3. Radiant Periodic: The pattern starts in B are *ordered monotonically.* We say that candidates of a pattern in a text are *ordered monotonically* if they are non-decreasing in both row and column indices or non-increasing in row index and non-decreasing in column index.

4. Lattice Periodic: The pattern starts in B fall on the nodes of a lattice. The lattice is defined by the basis vectors of the pattern (see [3, 9]).

(1) Non-periodic: If the pattern is non-periodic, then one candidate is killed in every duel. Since at the start there are at most $|compressed(T)|$ candidates, the number of duels is no more than $|compressed(T)|$.

(2,3) Line and Radiant Periodic: As in [4], we perform the duels within each logical column. Due to the transitivity lemma (lemma 3.1 in [4]) the number of duels within a logical column is no more than the number of candidates within the column. If the pattern is line or radiant periodic then at most 1 candidate remains alive in each column, resulting in at most m candidates. We move from right to left, adding one candidate at a time. Note that the transitivity lemma holds within the group of consistent candidates since they are ordered monotonically. Thus, the number of duels performed between columns is $O(m)$.

(4) Lattice Periodic: Similarly, in the lattice periodic case we first remove conflicts within each logical column. Moving from right to left, we add one candidate at time to the group of consistent candidates. A given column can contain several candidates, however, each new candidate can duel with **any** candidate to its right. If a candidate dies, then the operation is charged to the dead candidate. No more than $O(|compressed(T)|)$ such duels will take place. If both candidates remain alive, then the new candidate is consistent with all of the candidates to its right. We prove this in the following lemma. The lemma applies to a lattice periodic pattern, with basis vectors (r_1, c_1) and (r_2, c_2).

Lemma 4 *Given a set* Π *of compatible candidates in an* $m/2 \times m/2$ *text block, and given one additional candidate A within the same text block, if A is compatible with* **any** *one candidate in* Π*, then A is compatible with all candidates in* Π*.*

Proof: All candidates in Π lie on the nodes of a lattice defined by the basis vectors of the pattern. By definition of the lattice, the distance between *any* 2 compatible candidates is a linear combination of the basis vectors, $(ir_1 + jc_1, ir_2 + jc_2)$ proven in both [3, 9].

Let B be a token candidate from Π, and let B-A denote the distance between candidates B and A. We show that if A is compatible with B then A is compatible with every candidate C $\in \Pi$. If A and B are compatible, then B-A $= (ir_1 + jc_1, ir_2 + jc_2)$ for some integers i, j. For all C $\in \Pi$, B-C $= (pr_1 + qc_1, pr_2 + qc_2)$ for some integers p, q.

C-A = B-A - (B-C) $= ((i-p)r_1 + (j-q)c_1, (i-p)r_2 + (j-q)c_2)$, hence C is compatible with A. $\square$

4.2.3 Performing a Duel

The dueling paradigm is based on random access, both for retrieving the witness from the witness table, and for accessing the witness within the text. When the pattern and the text are compressed these elementary operations are no longer simple. Since the run-length compression is the concatenation of rows, accessing a relative row is always easy. However, the columns are difficult to pinpoint since the logical columns are hidden by the compression. In Section 2 we showed that provided with the proper pointers in the text we can access a witness in constant time. In this subsection we describe the complete duel which is performed in amortized $O(1)$ time.

We define 2 candidates UL (for upper left) and BR (for bottom right), such that $UL = (r_1, c_1, c'_1)$ and $BR = (r_2, c_2, c'_2)$. Initially we assume the knowledge of 2 critical pointers within the text.

1. In the row of BR (r_2) we have a pointer to the compressed location of the first logical column of UL (c'_1), called **beg_ptr**.
2. In the row of the witness (yet to be found out), we have a pointer to the compressed location of the last logical column of UL ($c'_1 + m$), called **end_ptr**.

The first step in performing a duel is to consult the witness table for the location of the witness. The witness table is indexed with the compressed distance between the 2 candidates involved in the duel. There are 2 problems that arise in this step. The first question is, what is the compressed distance between the 2 candidates? The row distance is simply $r_2 - r_1$, however, $c_2 - c_1$ has no meaning, since both numbers are affected by the run-lengths prior to the candidates. Second, it only makes sense to index the witness table with the compressed distance if the logical distance in the text is the same as in the pattern. Thus, after we find out the compressed distance we must check whether the compressed and uncompressed distances in the text match those of the pattern.

To answer the first question, we recall that we have a pointer, beg_pointer, that points to the logical column of UL in the row of BR. Therefore, we can simply subtract c_2-beg_ptr to obtain the compressed column distance. Let d denote this distance. The second step is to use the cumulative run-lengths of the pattern to determine whether the logical distance between UL and BR ($c'_2 - c'_1$) and the compressed distance (the answer to question 1) match the distances in the pattern. If the distances correspond, then we are ready to look up $Witness[r_2 - r_1 + 1, d + 1]$. Otherwise, candidate UL certainly contains an error, since in the row of BR the run-lengths differ from those of the pattern. Thus, we can kill candidate UL without checking any witness. In figure 5 we illustrate the two steps necessary to determine the witness location.

Remark: For case 2 patterns (see Algorithm A) beg_ptr is in row $r_2 + i$ where $(i, 0)$ of *compressed*(P) has length $< m/2$. The identical steps are performed using row $r_2 + i$ since the witness is stored at row $r_2 + i$.

Having retrieved the witness location, we can now randomly access the witness in the text in constant time. Recall that the witness location is stored as a compressed offset from the right edge of UL. The second pointer, end_ptr, points to the right edge of UL in the row of the witness. Thus, as proven in Lemma 2, we can offset from end_ptr to access the witness and the appropriate candidate will get killed. In figure 6 we show an example of a duel, illustrating why it is crucial to offset the witness from the end of the row.

It remains to show how we can have the 2 critical pointers, beg_ptr and end_ptr, in their proper places, without incurring a time penalty. Recall that the pointers point to the first and last column of UL, while the first pointer is in the row of BR and the second one is in the row of the witness. Since both of these rows are initially unknown, we need 2 pointers for *every* row of UL. Therefore, we maintain $3m$ pointers, 2 per row of the text. During each duel we calculate the positions of the pointers in the 2 pertinent rows, after BR and the witness are made known. We describe the way in which we handle the $3m/2$ pointers for the left edge of UL (beg_ptr at column c'_1), while the same logic applies for the remaining $3m/2$ pointers.

The dueling between columns begins at the right edge of the text. Thus, at the start of the dueling, the $3m/2$ pointers, one per row, are positioned at the right of the text. They are moved sequentially

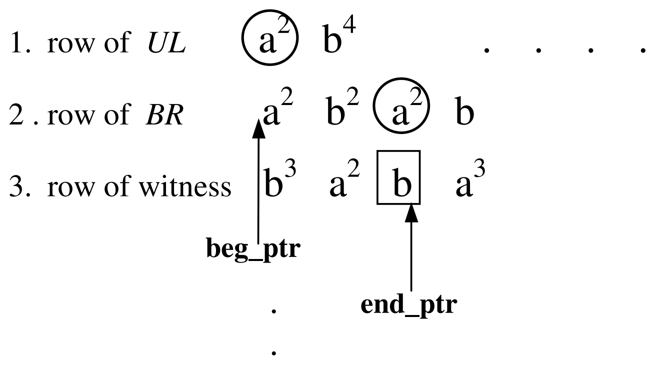


Figure 5: A sample text is shown, with 2 candidates marked, $UL = (1, 1, 1)$ and $BR = (2, 3, 5)$. The pattern being searched for is the one used throughout this paper. We first compute the compressed distance between UL and BR which is (1,2). If we would then go straight to the witness table, we would get $Witness[2, 3] = (3, -1)$. The 'b' at location (3,-1) would incorrectly kill candidate BR. Rather, using the cumulative run-lengths of the pattern, we first find that the uncompressed location of the candidate at compressed location (2,3) is (2,4), while in the text it is (2,5). This reveals that candidate UL is in error and no witness test is necessary.

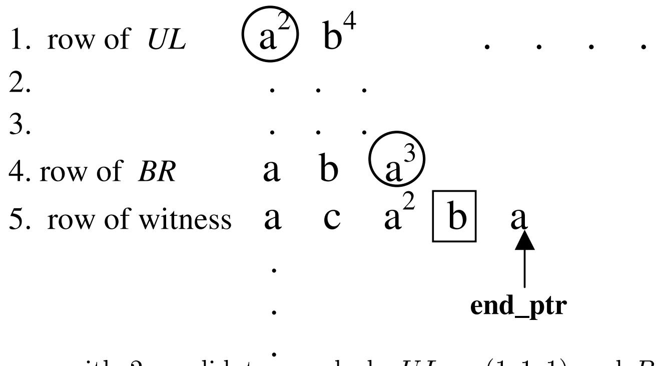


Figure 6: A sample text is shown, with 2 candidates marked. $UL = (1, 1, 1)$ and $BR = (4, 3, 4)$. The pattern being searched for is the one used throughout this paper. The first test reveals that the logical and compressed distances between UL and BR indeed coincide. We access $Witness[4, 3] = (5, -2)$. The 'b' at that location indicates that candidate UL should be killed. Note that if we would offset the witness from the left edge of UL, then the witness would be (5,2) and the incorrect candidate would be killed in this example.

towards the left (in a "comb-like" fashion), never further than the leftmost column involved in the duels. Whenever a pointer is moved, we calculate its position according to the run-lengths of the text characters. Since the duels are performed from right to left, at the start of a given duel all of the pointers must be either to the right of column c'_1 or at the location of c'_1. Only one pointer concerns a given duel, namely, the one in the row of BR. Therefore, we can simply check, in constant time per duel, whether the pointer in the row of BR is in the proper place. If it is not, then it is moved towards the left until it reaches the desired location.

Lemma 5 *The time complexity for the manipulation of the $3m/2$ pointers is $O(|compressed(T)|)$. Thus, the amortized cost of a duel is $O(1)$.*

Proof: There are 2 kinds of operations performed on the pointers. One is a move from a compressed text character to its neighbor on the left, and the second is a "move within a compressed character," where we find that the pointer is already at its proper location. Since the pointers are moved only towards the left, the number of times the first type of operation is executed can be no more than $|compressed(T)|$. The second type of operation is performed at most once per duel. As shown in the previous section, the number of duels is bound by the size of the compressed text. Therefore, the overall number of pointer moves is $O(|compressed(T)|)$, and each duel is done in amortized constant time. □

Algorithm B: Duel

Input: (1) 2 candidates, $UL = (c_1, r_1, c'_1)$ and $BR = (c_2, r_2, c'_2)$.
(2) 2 sets of $3m/2$ pointers (2 per row) $beg\_ptr[1 \ldots 3m/2]$, and $end\_ptr[1 \ldots 3m/2]$.
Output: Yes or no. If the answer is no, we output the candidate to be killed.

List of Variables:
rd=row distance
cd=compressed column distance
ucd=uncompressed column distance

Begin Algorithm

```
// Step B.1: Figure out the distance between the 2 candidates.
rd = r_2 - r_1
ucd = c'_2 - c'_1
move beg_ptr[r_2] to the left until logical column c'_1
cd = c_2 - beg_ptr[r_2]
// Step B.2: Check whether the distances correspond in the pattern.
If, in row rd of compressed(P), the position cd does not have run-length
   ucd, return no, killing UL
access Witness[rd+1,cd+1]
If the witness is a *, return yes.
// Step B.3: Access witness position in the compressed text.
move end_ptr, in the row of the witness, to the left until logical column c'_1 + m
```

offset the witness from the *end_ptr*, compare, and kill the appropriate candidate.

End Algorithm

4.3 Candidate Verification

Adapting the forward and backward waves for compressed matching is very complicated because the wave is strongly dependent upon the logical columns within the text. Furthermore, when working with a small text block of size $O(m^2)$ verification is much simplified. Therefore, in this section we introduce a new technique for verifying whether each candidate is an actual pattern occurrence. The standard verification entails comparing each text element with a given pattern element and marking the positions of each mismatch. Then, candidates that contain a mismatch within their domain are discarded. We show that we need not find every mismatch within the text, however, a mere $O(m)$ information suffices.

All candidates within an $m/2 \times m/2$ text square overlap, and thus it is sufficient to verify a given text row for 2 candidates: the rightmost and leftmost. In addition, it is not necessary to mark all of the mismatches in a given text row. We find the 2 mismatches in each row that are the closest to column $\frac{3}{4}m$, called the *center*. We then find the 2 closest mismatches to the center (one on each side) for each window of m rows. In Claim 1 we show that these 2 mismatches are enough to verify a given candidate.

Definition 4 (center) *We call the logical column $\frac{3}{4}m$ within the text the* center *of the text.*

Claim 1 *Given a set of mutually consistent candidates within a text of size $3m/2 \times 3m/2$, to verify whether a candidate in row r is a pattern occurrence it suffices to know exactly 2 mismatches in the window from row r to $r+m$: (1) the rightmost mismatch that occurs before the center, and (2) the leftmost mismatch that occurs after the center of the window.*

Proof: The proof is apparent from figure 7. If the rightmost mismatch in the entire window from row r to $r+m$, columns 1 to $3m/4$, does not fall within the domain of the candidate, then the candidate certainly does not contain any mismatches before the center. The same is true for columns $3m/4+1$ to $3m/2$. □

In order to find the mismatches closest to the center for all $m/2$ windows, we first find the mismatches closest to the center within each row. We then use the information about the individual rows to compute the mismatches for the windows.

4.3.1 Finding 2 mismatches per row

Our goal in this section is to find, for each text row, the 2 mismatches that are the closest to the center (one to the right and one to the left). We describe an algorithm that finds the mismatches closest to the center on the right side, and then show how the algorithm can be easily modified to find the mismatches closest to the center on the left side.

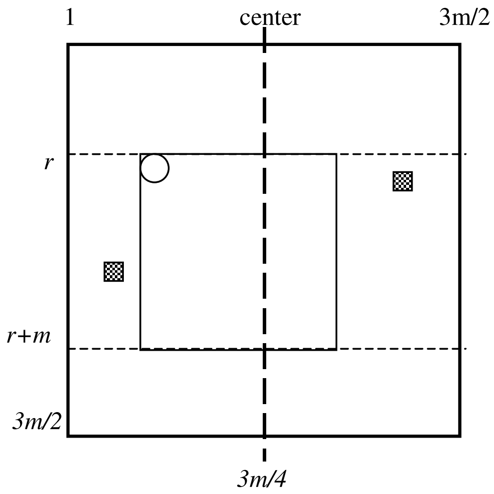


Figure 7: The small shaded rectangles mark the positions of the 2 mismatches within the rows r to $r+m$ that are the closest to the center, one on each side. The candidate pattern shown in row r does not include either of them, and is therefore a pattern occurrence.

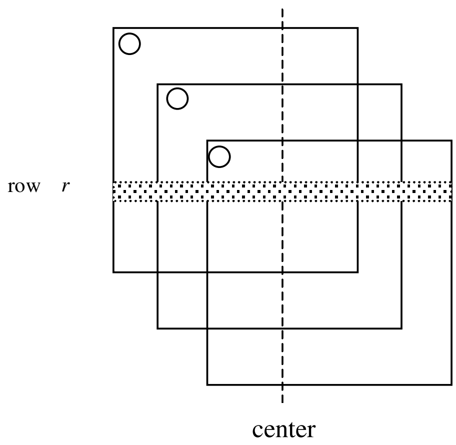


Figure 8: A sketch of 3 overlapping candidate patterns is shown. Consider a given row r that is included in all 3 candidates. It suffices to verify the area of row r that follows the center for the rightmost candidate alone while the area left of the center may be verified only for the leftmost candidate.

The input to this algorithm is a sorted list of consistent candidates (as shown previously in figure 4). All of the candidates lie in the upper left text square of (uncompressed) size $(\frac{m}{2})^2$. We begin processing the candidate list with the rightmost candidate. The rightmost candidate begins before the center (as do all of the candidates), and it ends the furthest to the right of all candidates (see figure 8). Recall that all overlapping candidates expect the same elements in the overlapping region. The region that we are dealing with is the one past the center, from column $3m/4 + 1$ to $3m/2$. Therefore, once we verify the area of the rightmost candidate that follows the center, it is not necessary to visit any row included in the rightmost candidate again.

We verify the rightmost candidate in the brute-force way. We begin at the left, and move along the text (doing nothing) until we reach the center. We then compare the text elements to the appropriate pattern elements until a mismatch is found, or the end of the pattern row is reached. If there is more than one candidate in the same logical column, then verification of the rows is done in a wave-like fashion, beginning again with row 1 for each additional candidate. Note that all candidates within the same logical column must overlap. Therefore, upon completing the verification of the candidates in the rightmost column, an interval of at least m text rows have been processed. These rows will not be visited again.

The algorithm continues with the next column to the left that contains one or more candidates. The interval of text rows included in these candidates is either subsumed in the scanned interval, in which case no verification is necessary, or it extends the scanned interval from above and/or below. In the latter case, we must verify the additional rows in the same manner previously described. Thus, we move towards the left, considering each candidate and verifying those rows that have not yet been verified.

This algorithm can be easily modified to find the mismatches closest to the center on the left side. In this case, the leftmost candidate begins the first and ends at or past the center (see figure 8). Since the candidates are linked from left to right, we can perform the same algorithm beginning with the leftmost candidate and moving towards the right.

Complexity: The time complexity for this step is $O(|compressed(P)| + |compressed(T)|)$. For each direction every candidate is processed once, and there are at most $|compressed(P)|$ candidates. Over all candidates, each text row is visited at most once. Verifying a text row consists of constant time comparisons of its elements with the elements of the compressed pattern.

4.3.2 Finding 2 mismatches per window

We now show how to transform the information computed in the previous section into a more useful format.
Given: The location of the first mismatch following the center for *each text row*.
Compute: The location of the first mismatch following the center for *each window of m text rows.*

The problem can be stated in simpler terms as follows. Given an array of $3m/2$ numbers, find the minimum of each (sliding) window of size m in $O(m)$ time (see figure 9). Although this problem is not trivial, we exploit a property of the given numbers, allowing us to solve it in $O(m)$ time. The numbers stored in our array range from $3m/4 + 1$ to $3m/2$. Initially, we bucket sort the array of numbers. We then consider each number, beginning with the minimum. Let q be the row of the minimum of the entire array. If $q \leq m$ then it is also the minimum of all windows beginning from 1 to q. Similarly, if $q > m$ then it plays as the minimum of all windows from $q - m$ to $m/2$.

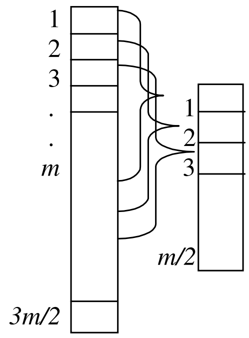


Figure 9: The left array of size $3m/2$ holds the first mismatch following the center for each row of the text. This information must be transformed into the smaller array of size $m/2$, where each element represents the first mismatch to follow the center for each *window* of m text rows.

Thus, at each step we maintain 2 intervals, one beginning at 1, and the second ending at $m/2$. The values of all entries in both intervals are known. Consider the number that is the second to the minimum. Using the same idea as in the previous section, the new number is either included in one of the known intervals, or it extends an interval by setting the values of new windows. The algorithm continues in this manner, considering the next minimum value, until the two intervals merge into one. Algorithm C implements the transformation. The algorithm uses 2 pointers, *low* and *high*, such that at every point the values for all windows from 1 to *low*, as well as from *high* to $m/2$ are known.

Algorithm C: Transform mismatches per row to mismatches per window

Input: array row_mismatch$[1 \dots 3m/2]$, where row_mismatch$[i]$ is the logical column of the first mismatch in row i following the center.
Output: array window$[1 \dots m/2]$, where window$[i]$ is the first mismatch following the center in the entire window extending from row i to $i + m$.

Begin Algorithm

Step C.1: Bucket sort the numbers in the array row_mismatch into $3m/4$ buckets.
Each bucket contains a linked list of elements. An element is denoted by a pair (row, col) where column col is the position of the first mismatch in row row following the center.

Step C.2: $low = 1$
$high = m/2$
for $i = 3m/4 + 1$ to $3m/2$ do
  while bucket[i] is not empty AND $low < high$
    $min =$ the next element in bucket[i]
    //if the minimum is in the *upper* m rows, then it is the
    // minimum of all windows that begin higher than it.
    if $min.row \leq m$
      for $j = low$ to $min.row$

```
            if (j > high) break;
            window[j]=min.col
        end for
        low = min.row
    // if the minimum is in the lower m/2 rows (between m and 3m/2),
    // then it is the minimum of all windows that end below it.
    else if min.row > m
        for j = high down to min.row − m
            if (j ≤ low) break;
            window[j]=min.col
        end for
        high = min.row − m
    end if
  end while
end for
```

End Algorithm

Complexity: The time complexity of Algorithm C is $O(m)$. The bucket sort is linear in the size of the given array. We process each value in the array, and over all values we do not set more than $m/2$ locations in the output array.

5 Trivial Rows

Patterns that contain trivial rows present difficulties in compressed matching since many overlapping patterns may start at the same compressed text location (a contradiction to Observation 1). In fact, if all of the pattern rows are trivial then the size of the output may be larger than the size of the compressed text. Consequently, the ideas presented in this paper cannot be directly applied to patterns with trivial rows. However, we can modify the algorithm to find all patterns if we allow time $O(|T|)$. The new algorithm still has extra space $O(|compressed(P)|)$.

The main difficulty arises in the first stage, described in Section 4.1. At the start of the algorithm, we divide the text into small blocks of size $3m/2 \times 3m/2$. Runs with length greater than m are skipped. If the pattern contains trivial rows, then we cannot skip all areas of the text that contain runs longer than m. To obtain an algorithm with time complexity $O(|compressed(T)|)$ the entire text must be processed at once. However, if we allow time $O(|T|)$, then we can divide the text and apply the ideas developed in this paper.

It should be noted that even the "simple" case where *all* pattern rows are trivial is complicated. The problem of pattern matching in dynamic texts and fixed pattern can be reduced to it. The latter problem is interesting and difficult. The authors are preparing a different paper for its solution, and its time complexity is not linear.

In this section we give a brief description of the algorithm which allows trivial rows in the pattern. We deal separately with 2 types of patterns.

1. At least one row of the pattern in non-trivial.
2. All rows of the pattern are trivial.

5.1 Patterns with trivial and non-trivial rows

If the pattern contains at least 1 non-trivial row then the algorithm presented in this paper can be used with the following modifications.

5.1.1 Witness Table Construction

A witness table is built for the compressed pattern, as described in Section 3.1. We show, as in Lemma 1 of Section 3.1, that its size is $O(|compressed(P)|)$. Cases 1 and 2 of Lemma 1 apply for patterns with trivial rows. Case 3 differs since a trivial row is both the first and last compressed character in its row. However, we can still build a witness table of size $O(|compressed(P)|)$. The intuition is that as long as there is some row of the pattern that is non-trivial, then it will serve as the implicit witness for most locations. For all locations for which it is not a witness, the overlapping pattern matches at the splitpoint, i.e. the pattern has an additional compressed character. We replace Case 3 with the following.

Case 3: $compressed(P)[i, 0] = a^k | k \geq m/2$ for all rows $0 \leq i < m/2$

Label the quadrants of P counterclockwise 1,2,3 and 4. As in case 2 (see Section 3.1), by symmetry, any split in quadrant 3 can be used as an implicit witness. If there is no split in quadrant 1 or 3 then any split in quadrant 4 can be used as an implicit witness for all but one location. We use the split in quadrant 4 that is closest to row $m/2$. Similarly, the rightmost split in quadrant 2 is an implicit witness for all but one location.

5.1.2 Marking Potential Candidates

The text is divided into $3m/2 \times 3m/2$ size blocks. Note that we cannot skip long runs, and hence this division results in $O(|T|)$ work. The algorithm for marking potential candidates, described in Section 4.2.1, cannot be used. However, we use a similar idea to achieve the same goal. Instead of searching for a given pattern row, we search for the 2d aperiodic root of the pattern. Each occurrence of this root defines at most one possible pattern start and there can be at most $|compressed(P)|$ occurrences in a $3m/2 \times 3m/2$ text block. An aperiodic 2d pattern can be found using the previous algorithm.

5.1.3 Candidate Consistency and Verification

The dueling and verification work exactly as they do for the patterns with non-trivial rows.

5.2 Patterns with all trivial rows

If all of the rows of the pattern are trivial, then we can view the compressed pattern as a 1-D string. The preprocessing of the pattern consists of building a KMP automaton for the characters of $compressed(P)$. The text scanning algorithm works as follows.

1. Divide the text into overlapping regions of size $2m - 1 \times 2m - 1$.
2. Mark off the runs in each row with length $>= m$ beginning in columns $1 \ldots m - 1$.
3. Run the KMP algorithm down the columns matching the character only.
4. Line up the left and right intervals properly.

Note that when dividing the text (step 1), we can skip all runs that have length $< m$. However, the worst case runtime is still $O(|T|)$.

6 Conclusion

In this paper we introduced a new complexity measure for compressed matching. A compressed matching is called *inplace* if the extra space used is proportional to the input size of the pattern. The goal of finding an inplace algorithm for various compressions opens new avenues of research in the area of compressed matching. The first inplace compressed matching algorithm, presented in this paper, solves the 2D run-length compressed matching problem. Run-length compression is used in FAX transmission. Other interesting open problems are to find inplace algorithms to solve the one and two-dimensional compressed matching problems for the LZ1, LZ2 and LZW compression techniques.

References

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279–288, Mar 1992.

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL*, pages 440–452, Jan 1992.

[3] A. Amir and G. Benson. Two-dimensional periodicity and its application. *SIAM J. Comp.*, 27(1):90–106, February 1998.

[4] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

[5] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Journal of Algorithms*, 24(2):354–379, August 1997.

[6] A. Amir, G. Landau, and D. Sokol. Inplace run-length 2d compressed search. In *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 817–818, 2000.

[7] P. Berman, M. Karpinski, L. Larmore, W. Plandowski, and W. Rytter. On the complexity of pattern matching for highly compressed two dimensional texts. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM 97)*, pages 40–51. LNCS 1264, Springer, 1997.

[8] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. *Proc. 27th Annual ACM Symposium on the Theory of Computing*, pages 703–712, 1995.

[9] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Comp.*, 25(5):907–935, October 1996.

[10] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Randomized efficient algorithms for compressed strings: The finger-print approach. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, pages 39–49. LNCS 1075, Springer, 1996.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architeture: A Quantitative Approach.* Morgan Kaufmann, 2 edition, 1996.

[12] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[13] M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. of Algorithms*, pages 422–432, 1984.

[14] U. Vishkin. Optimal parallel pattern matching in strings. *Proc. 12th ICALP*, pages 91–113, 1985.

[15] J. Ziv. personal communication. 1995.